



Ho Chi Minh City University of Technology  
Faculty of Computer Science and Engineering



# Data Structures and Algorithms – C++ Implementation

---

TÀI LIỆU SƯU TẬP  
Huỳnh Tấn Đạt

Email: [htdat@cse.hcmut.edu.vn](mailto:htdat@cse.hcmut.edu.vn)

Home Page: <http://www.cse.hcmut.edu.vn/~htdat/>

# Pointer in C++

- ❑ Declaration

Node \*ptr;

- ❑ Create an object

ptr = new Node();

- ❑ A pointer usage

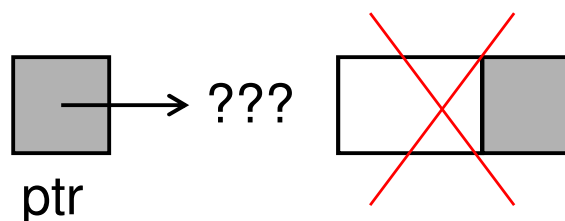
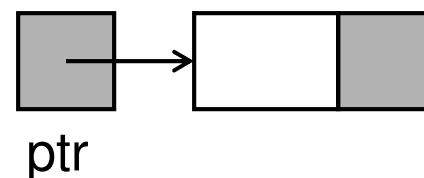
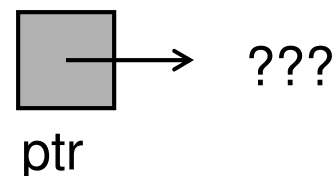
printf("Data in node: %d", ptr->data);

- ❑ Destroy an object

delete ptr;

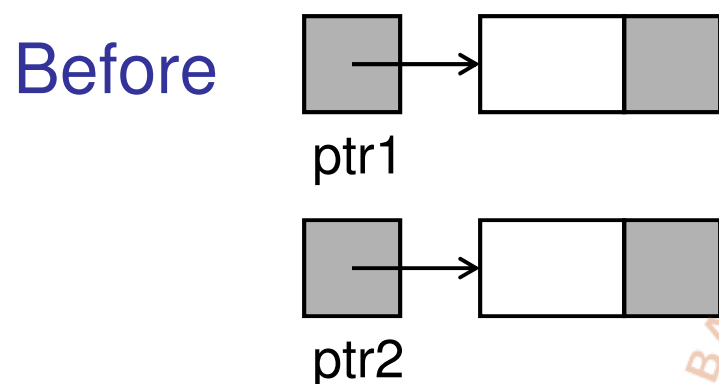
- ❑ NULL pointer

ptr = NULL;



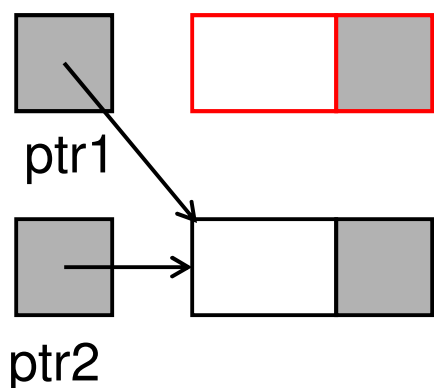
# Pointer in C++

❑ Be careful in these cases:

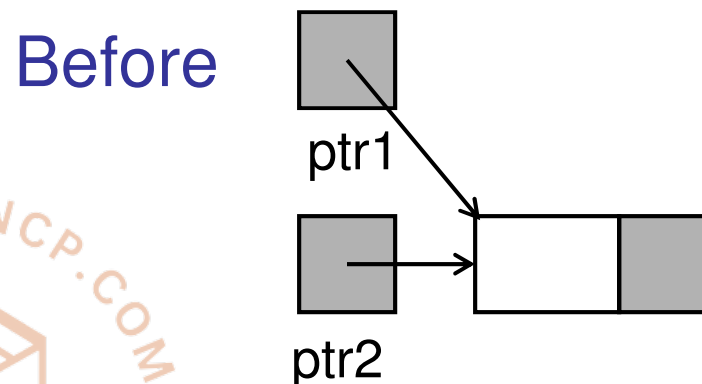


`ptr1 = ptr2;`

After

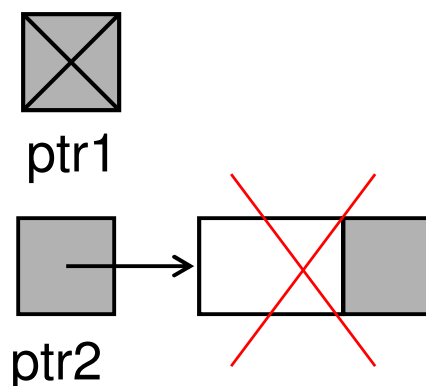


Garbage



`delete ptr1; ptr1 = NULL;`

After



Dangling reference problem

# Parameter Passing Techniques

```
void func(int* a, int* b){
```

```
    int *t;
```

```
    t = a;
```

```
    a = b;
```

```
    b = t;
```

```
}
```

```
void func(int* &a, int* &b){
```

```
    int *t;
```

```
    t = a;
```

```
    a = b;
```

```
    b = t;
```

```
}
```

```
void main() {
```

```
    int *p1 = new int;
```

```
    *p1 = 10;
```

```
    int *p2 = new int;
```

```
    *p2 = 20;
```

```
    func(p1, p2);
```

```
    printf("%d", *p1);
```

```
    printf("%d", *p2);
```

# Parameter Passing Techniques

```
void func(int* &a, int* b){
```

```
    int *t;
```

```
    t = a;
```

```
    a = b;
```

```
    b = t;
```

```
}
```

```
void func(int* a, int* &b){
```

```
    int *t;
```

```
    t = a;
```

```
    a = b;
```

```
    b = t;
```

```
}
```

```
void main() {
```

```
    int *p1 = new int;
```

```
    *p1 = 10;
```

```
    int *p2 = new int;
```

```
    *p2 = 20;
```

```
    func(p1, p2);
```

```
    printf("%d", *p1);
```

```
    printf("%d", *p2);
```



# Parameter Passing Techniques

---

```
void func(int **a, int **b){  
    int *t;  
    t = *a;  
    *a = *b;  
    *b = t;  
}
```

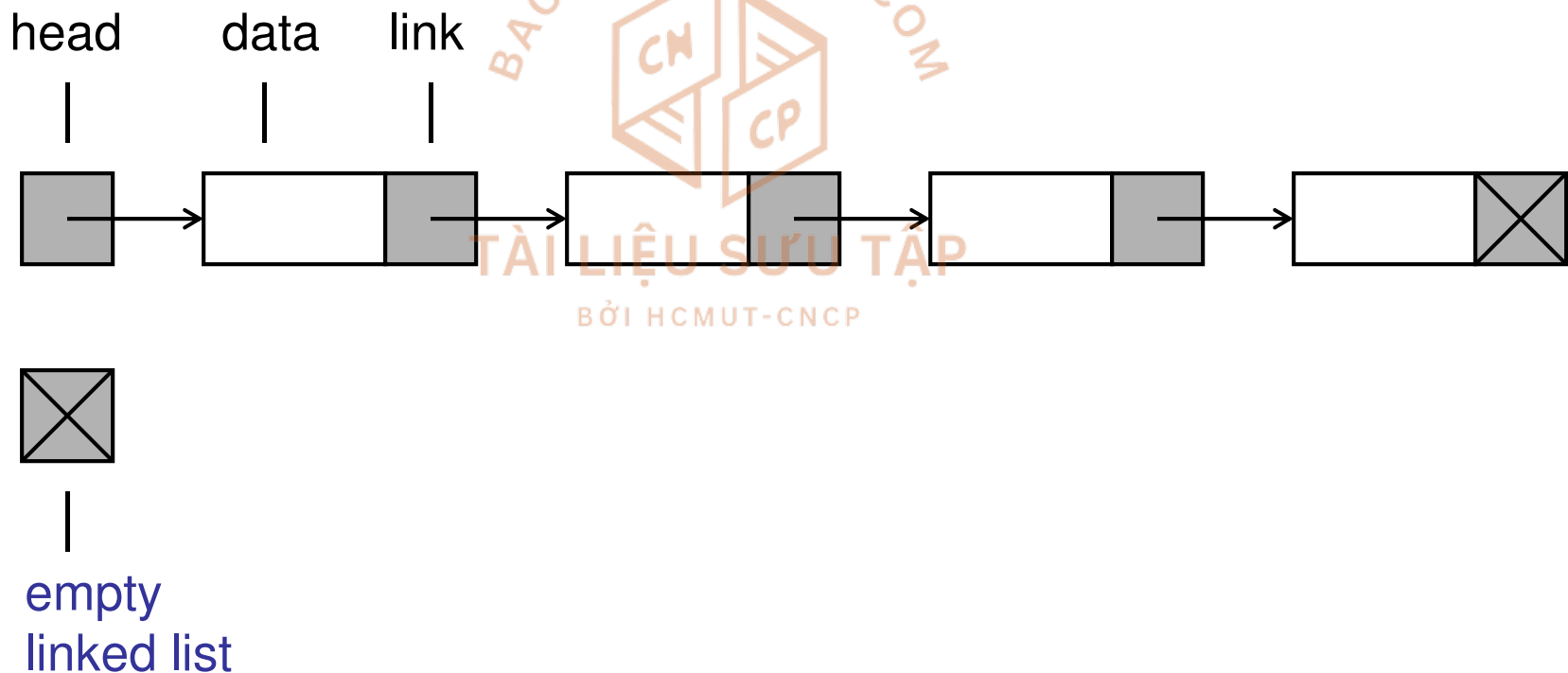
```
void main() {  
    int *p1 = new int;  
    *p1 = 10;  
    int *p2 = new int;  
    *p2 = 20;  
    func(&p1, &p2);  
    printf("%d", *p1);  
    printf("%d", *p2);  
}
```



# Linked Lists

- ❑ A linked list is an ordered collection of data in which each element contains the location of the next element

Element = Data + Link



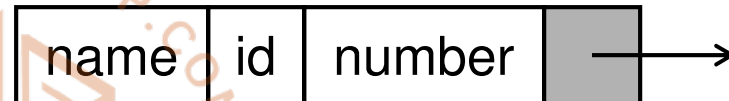
# Nodes

---

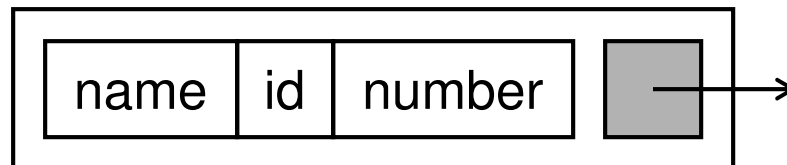
A node with  
one data field



A node with  
three data fields

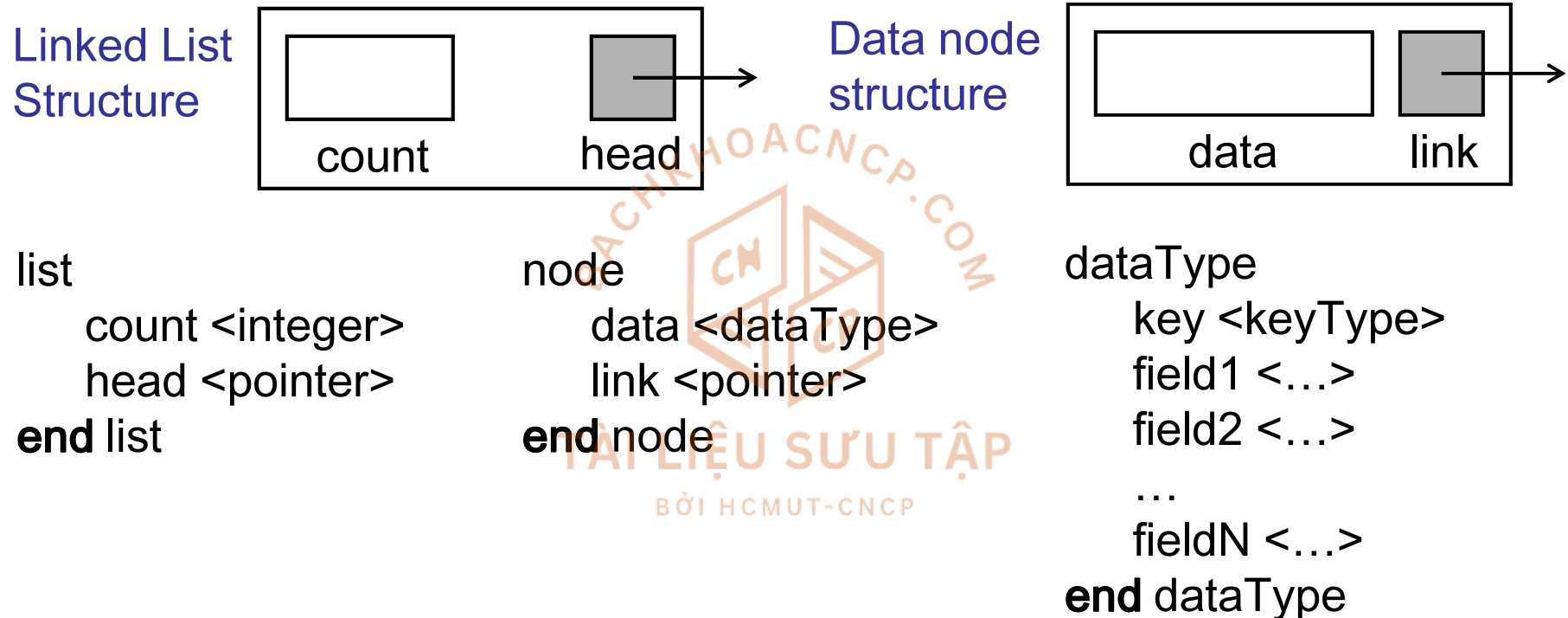


A node with one  
structured data field





# Nodes



# Nodes – Implementation in C++

---

```
struct Node {  
    int data;  
    Node *next;  
};
```

```
node  
    data <dataType>  
    link <pointer>  
end node
```



# Nodes – Implementation in C++

```
Node *p = new Node();
```

```
p->data = 5;
```

```
cout<< p->data;
```

```
Node *q = p;
```

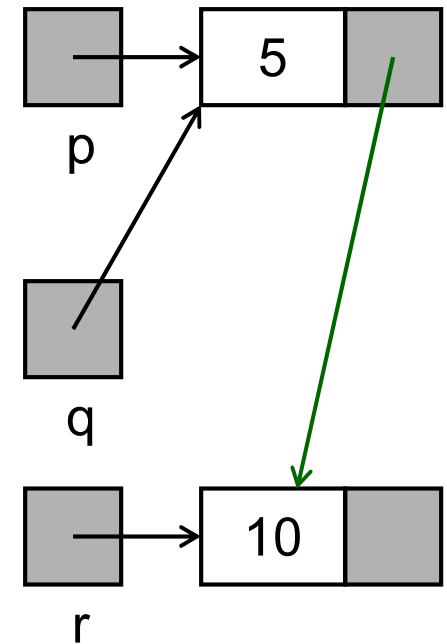
```
cout<< q->data;
```

```
Node *r = new Node();
```

```
r->data = 10;
```

```
q->next = r;
```

```
cout<< p->next->data;
```



# Nodes – Implementation in C++

---

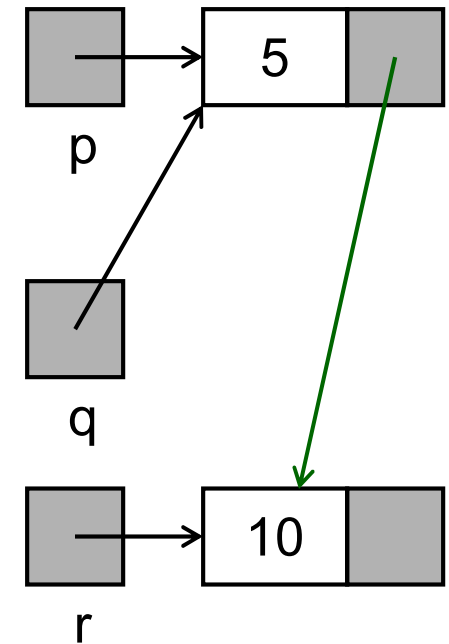
```
struct Node {  
    int data;  
    Node *next;  
};
```

```
struct Node {  
    float data;  
    Node *next;  
};
```

```
template <class ItemType>  
struct Node {  
    ItemType data;  
    Node<ItemType> *next;  
};
```

# Nodes – Implementation in C++

```
Node<int> *p = new Node<int>();  
p->data = 5;  
cout<< p->data;  
Node<int> *q = p;  
cout<< q->data;  
Node<int> *r = new Node<int>();  
r->data = 10;  
q->next = r;  
cout<< p->next->data;
```



# Nodes – Implementation in C++

---

```
template <class ItemType>
class Node{
public:
    Node() {
        this->next = NULL;
    }
    Node(ItemType data) {
        this->data = data;
        this->next = NULL;
    }

    ItemType data;
    Node<ItemType> *next;
};
```

# Linked List – Implementation in C++

---

```
template <class List_ItemType>
class LinkedList{
public:
    LinkedList();
    ~LinkedList();
protected:
    Node<List_ItemType>* head;
    int count;
};
```

```
list
    count <integer>
    head <pointer>
end list
```



# Linked List Algorithms

---

- ☐ Create list
- ☐ Insert node
- ☐ Delete node
- ☐ Traverse
- ☐ Destroy list





# Linked List Implementation

---

```
template <class List_ItemType>
class LinkedList{
public:
    LinkedList();
    ~LinkedList();
protected:
    int InsertNode(Node<List_ItemType>* pPre,
                  List_ItemType value);
    List_ItemType DeleteNode(Node<List_ItemType>* pPre,
                             Node<List_ItemType>* pLoc);
    int Search(List_ItemType value, Node<List_ItemType>*
    &pPre, Node<List_ItemType>* &pLoc);
    void Traverse();

    Node<List_ItemType>* head;
    int count;
};
```

# Linked List Implementation

---

```
template <class List_ItemType>
class LinkedList{
public:
    LinkedList();
    ~LinkedList();
    void InsertFirst(List_ItemType value);
    void InsertLast(List_ItemType value);
    int InsertItem(List_ItemType value, int Position);
    List_ItemType DeleteFirst();
    List_ItemType DeleteLast();
    int DeleteItem(int Position);
    int GetItem(int Position, List_ItemType &dataOut);
    void Print2Console();
    void Clear();
    // Augment your methods for linked list here!!!
    LinkedList<List_ItemType>* Clone();
protected:
    // ...
```

# Linked List Implementation

---

## ❑ How to use Linked List data structure?

```
int main(int argc, char* argv[]) {  
    LinkedList<int>* myList =  
        new LinkedList<int>();  
    myList->InsertFirst(15);  
    myList->InsertFirst(10);  
    myList->InsertFirst(5);  
    myList->InsertItem(18, 3);  
    myList->InsertLast(25);  
    myList->InsertItem(20, 3);  
    myList->DeleteItem(2);  
    printf("List 1:\n");  
    myList->Print2Console();  
}
```

# Linked List Implementation

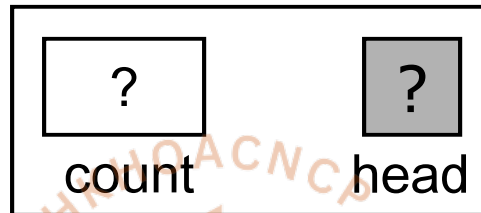
---

```
// ...  
int value;  
LinkedList<int>* myList2 = myList->Clone();  
printf("\nList 2:\n");  
myList2->Print2Console();  
myList2->GetItem(1, value);  
printf("Value at position 1: %d", value);  
  
delete myList;  
delete myList2;  
return 1;  
}
```

# Create List

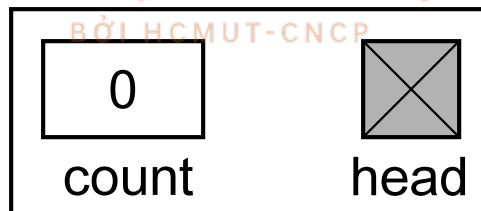
---

Before list



list.head = null  
list.count = 0

After list



# Create List

---

**Algorithm** createList (**ref** list <metadata>)

Initializes metadata for a linked list

**Pre** list is a metadata structure passed by reference

**Post** metadata initialized

1 list.head = null

2 list.count = 0

3 return

**End** createList



# Linked List Implementation

---

```
template <class List_ItemType>
LinkedList<List_ItemType>::LinkedList() {
    this->head = NULL;
    this->count = 0;
}
```



# Insert Node

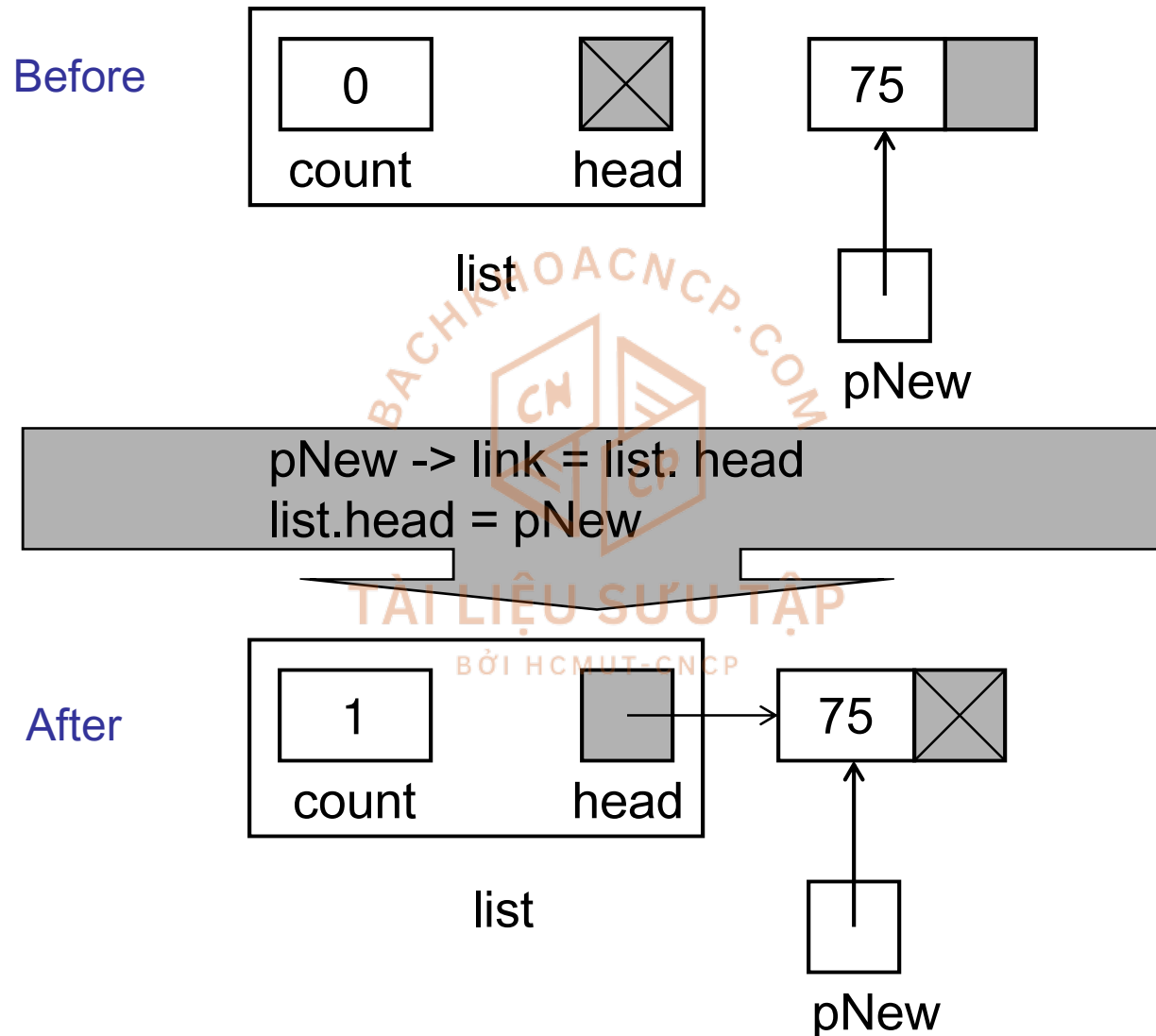
---

- ❑ Allocate memory for the new node and set up data
- ❑ Point the new node to its successor
- ❑ Point the new node's predecessor to it



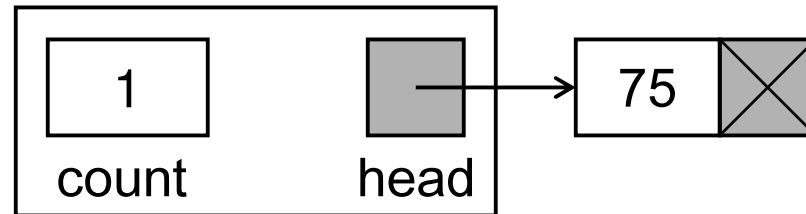


# Insert into Empty List

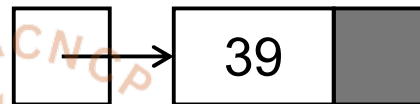


# Insert at the Beginning

Before



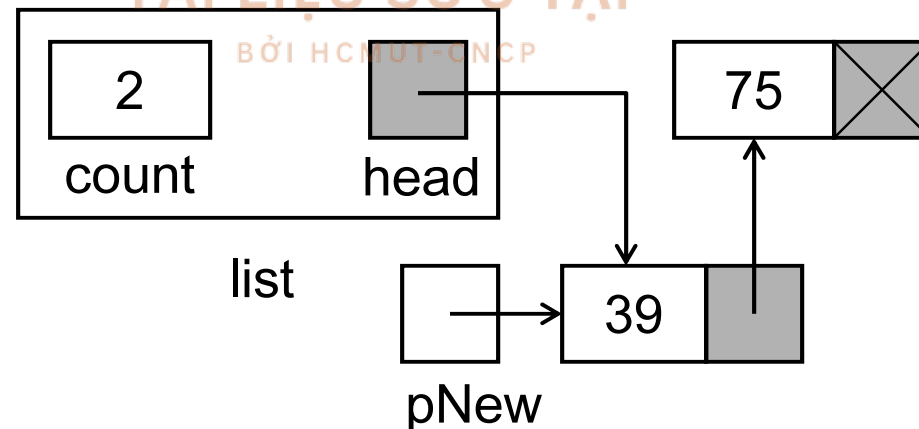
list



pNew

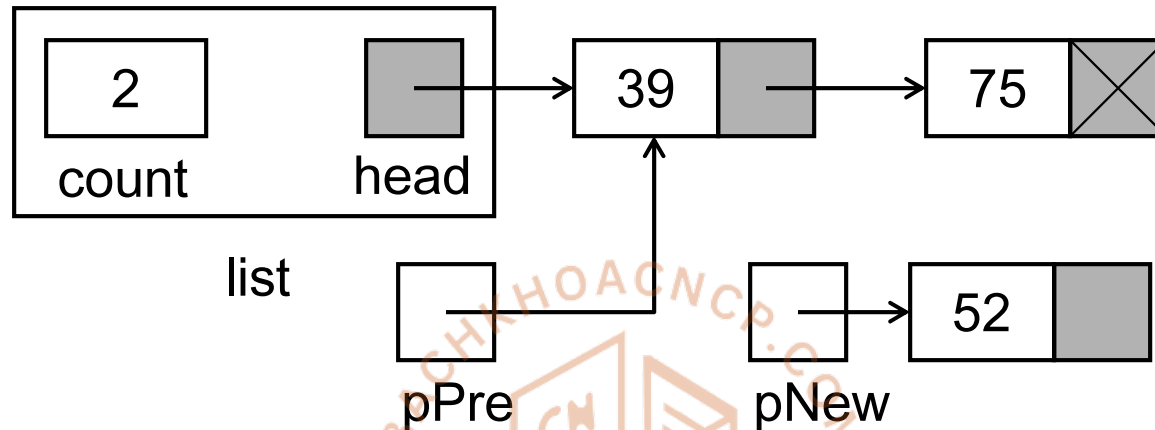
pNew -> link = list.head  
list.head = pNew

After



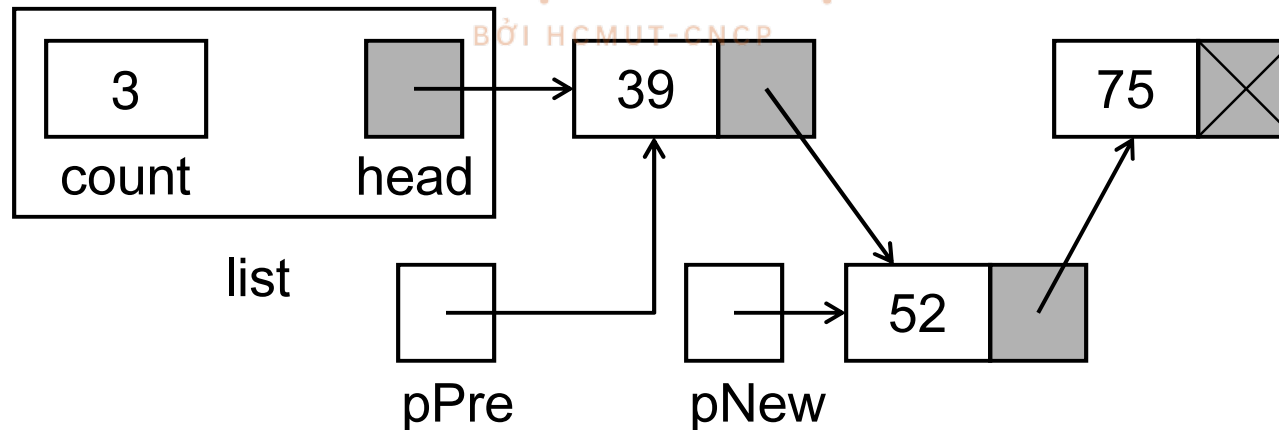
# Insert in Middle

Before



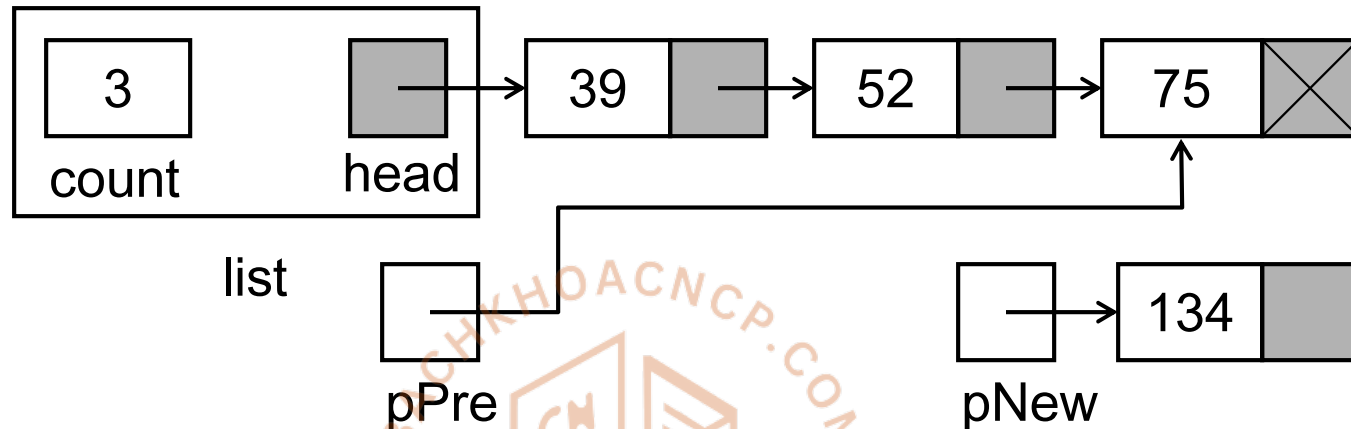
$pNew \rightarrow link = pPre \rightarrow link$   
 $pPre \rightarrow link = pNew$

After



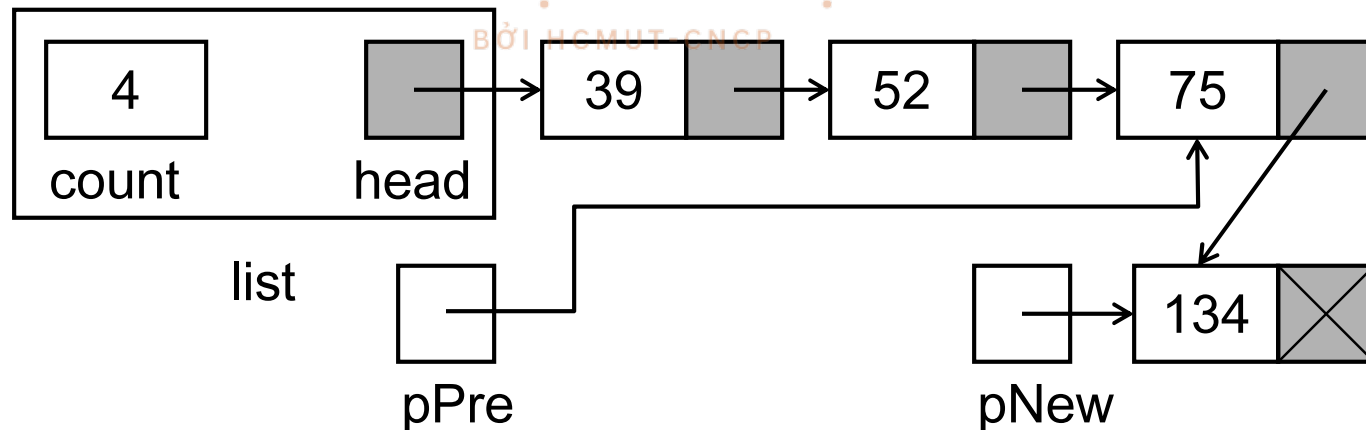
# Insert at End

Before



$pNew \rightarrow link = pPre \rightarrow link$   
 $pPre \rightarrow link = pNew$

After



# Insert Node Algorithm

---

**Algorithm** insertNode (ref **list** <metadata>,  
                                  val **pPre** <node pointer>,  
                                  val **dataIn** <dataType>)

Inserts data into a new node in the linked list

**Pre**     **list** is metadata structure to a valid list  
          **pPre** is pointer data's logical predecessor  
          **dataIn** contains data to be inserted

**Post**    data have been inserted in sequence

**Return** true if successful, false if memory overflow

# Insert Node Algorithm

---

```
1  allocate(pNew)
2  if (memory overflow)
    1  return false
3  pNew -> data = dataIn
4  if (pPre = null)
    Adding before first node or to empty list
    1  pNew -> link = list.head
    2  list.head = pNew
5  else
    Adding in middle or at end
    1  pNew -> link = pPre -> link
    2  pPre -> link = pNew
6  list.count = list.count + 1
7  return true
End    insertNode
```

# Insert Node

```
template <class List_ItemType>
int LinkedList<List_ItemType>::InsertNode(
    Node<List_ItemType> *pPre, List_ItemType value) {
    Node<List_ItemType> *pNew = new Node<List_ItemType>();
    if (pNew == NULL)
        return 0;
    pNew->data = value;
    if (pPre == NULL){
        pNew->next = this->head;
        this->head = pNew;
    } else {
        pNew->next = pPre->next;
        pPre->next = pNew;
    }
    this->count++;
    return 1;
}
```

# Delete Node

---

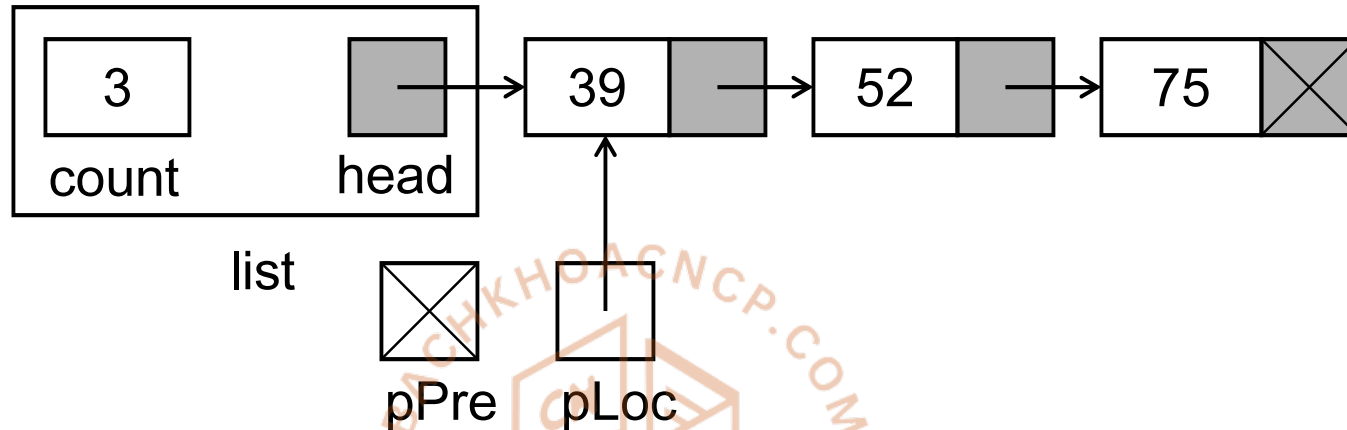
- ☐ Locate the node to be deleted.
- ☐ Point the node predecessor's link to its successor.
- ☐ Release the memory for the deleted node





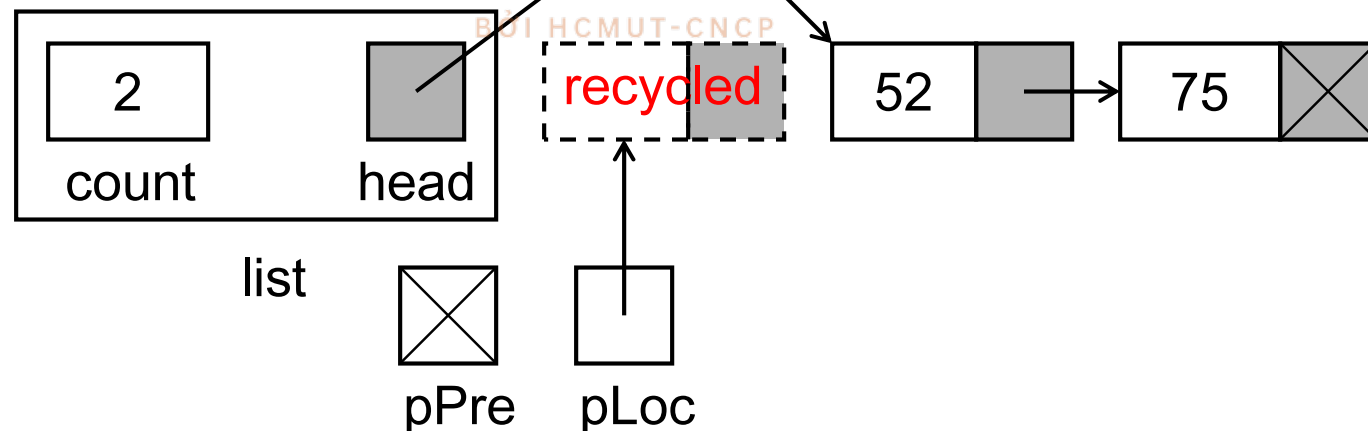
# Delete First Node

Before

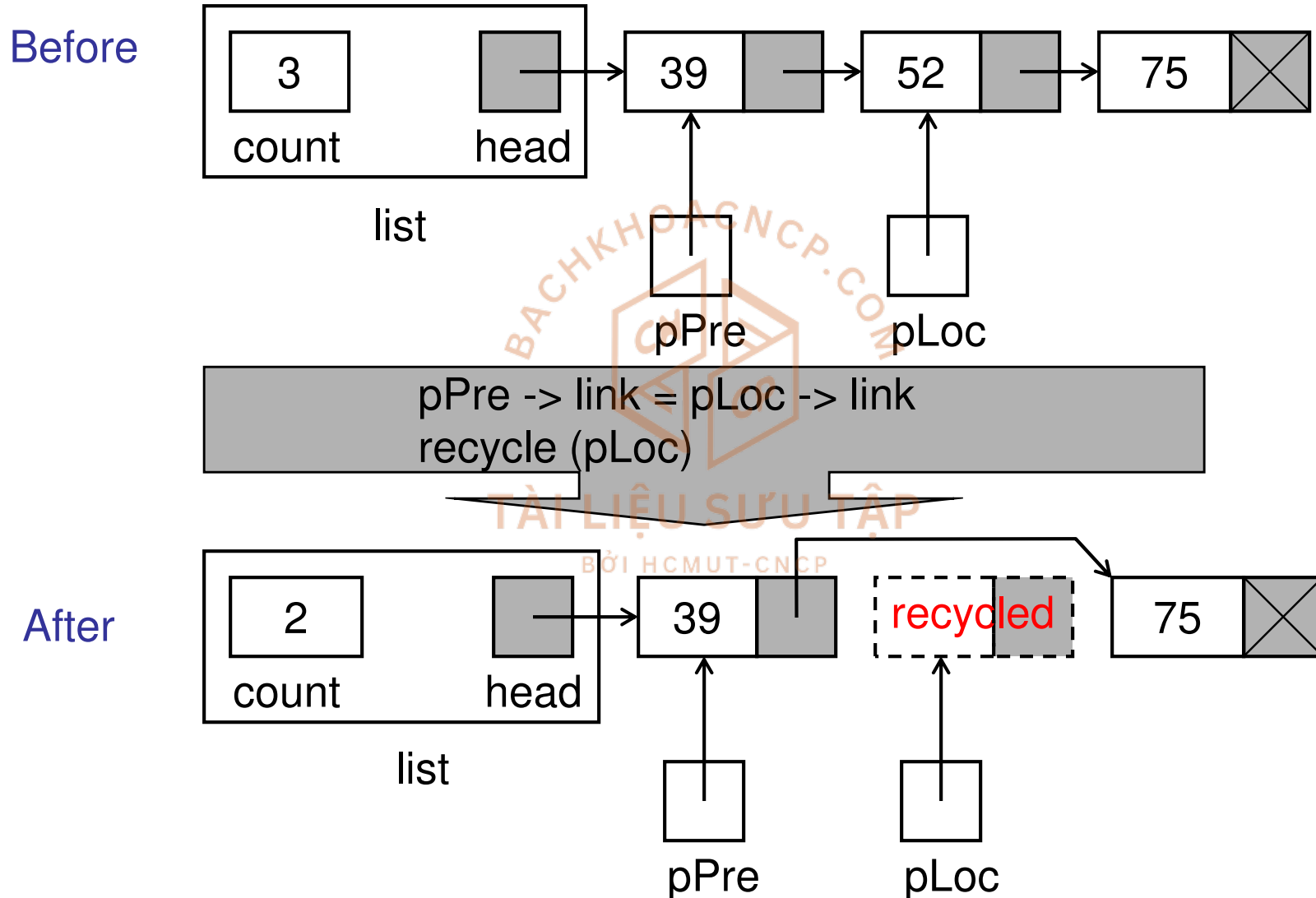


list.head = pLoc -> link  
recycle(pLoc)

After



# General Delete Case



# Delete Node Algorithm

---

**Algorithm** deleteNode (**ref** list <metadata>,  
                          val pPre <node pointer>,  
                          val pLoc <node pointer>  
                          **ref** dataOut <dataType>)

Delete data from a linked list and returns it to calling module

**Pre** list is metadata structure to a valid list  
     pPre is a pointer to predecessor node  
     pLoc is a pointer to node to be deleted  
     dataOut is variable to receive deleted data

**Post** data have been deleted and returned to caller

# Delete Node Algorithm

---

1 dataOut = pLoc -> data

2 if (pPre = null)

## **Delete first node**

1 list.head = pLoc -> link

3 else

## **Delete other nodes**

1 pPre -> link = pLoc -> link

4 list.count = list.count - 1

5 recycle (pLoc)


6 return

**End** deleteNode

# Delete Node

---

```
template <class List_ItemType>
List_ItemType LinkedList<List_ItemType>::DeleteNode(
    Node<List_ItemType> *pPre, Node<List_ItemType> *pLoc) {
    List_ItemType result = pLoc->data;
    if (pPre == NULL)
        list->head = pLoc->next;
    else
        pPre->next = pLoc->next;
    this->count--;
    delete pLoc;
    return result;
}
```



# Traverse List

---

- ❑ Traverse module controls the loop:  
calling a user-supplied algorithm to process data

pWalker = list.head

loop (pWalker not null)

process (pWalker -> data)

pWalker = pWalker -> link

# Traverse List


```
template <class List_ItemType>
void LinkedList<List_ItemType>::Traverse() {
    Node<List_ItemType> *p = head;
    while (p != NULL) {
        p->data++; // process data here!!!
        p = p->next;
    }
}
```

```
template <class List_ItemType>
void LinkedList<List_ItemType>::Traverse(void
    (*visit)(List_ItemType &)) {
    Node<List_ItemType> *p = head;
    while (p != NULL) {
        (*visit)(p->data);
        p = p->next;
    }
}
```

# Searching in Linked List

---

```
template <class List_ItemType>
int LinkedList<List_ItemType>::
    Search(List_ItemType value, Node<List_ItemType>* &pPre,
    Node<List_ItemType>* &pLoc) {
    pPre = NULL;
    pLoc = this->head;
    while (pLoc != NULL && pLoc->data != value) {
        pPre = pLoc;
        pLoc = pLoc->next;
    }
    return (pLoc != NULL); // found: 1; notfound: 0
}
```

A watermark logo is centered over the code. It consists of a diamond shape divided into four quadrants. The top-left quadrant contains the letters 'CN', the top-right contains 'CP', the bottom-left contains 'H', and the bottom-right contains 'M'. Above the diamond, the text 'BACHKHOACNCP.COM' is written in a curved path. Below the diamond, the text 'TÀI LIỆU SƯU TẬP' is written in a straight line, and below that, 'BỞI HCMUT-CNCP' is written in a smaller font.



# Destroy List Algorithm

---

**Algorithm** destroyList (val list <metadata>)

Deletes all data in list.

**Pre** list is metadata structure to a valid list

**Post** all data deleted

- 1 loop (list.head not null)
  - 1 dltPtr = list.head
  - 2 list.head = this.head -> link
  - 3 recycle (dltPtr)

**No data left in list. Reset metadata**

2 list.count = 0

3 return

**End** destroyList

# Destroy list

```
template <class List_ItemType>
void LinkedList<List_ItemType>::Clear() {
    Node<List_ItemType> *temp;
    while (this->head != NULL) {
        temp = this->head;
        this->head = this->head->next;
        delete temp;
    }
    this->count = 0;
}

template <class List_ItemType>
LinkedList<List_ItemType>::~~LinkedList() {
    this->Clear();
}
```

# Exercises

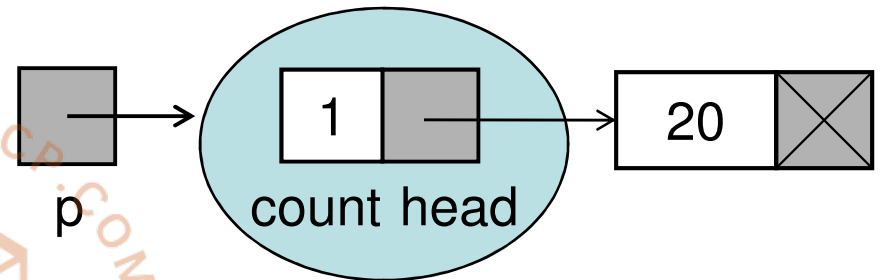
---

```
template <class List_ItemType>
class LinkedList{
public:
    LinkedList();
    ~LinkedList();

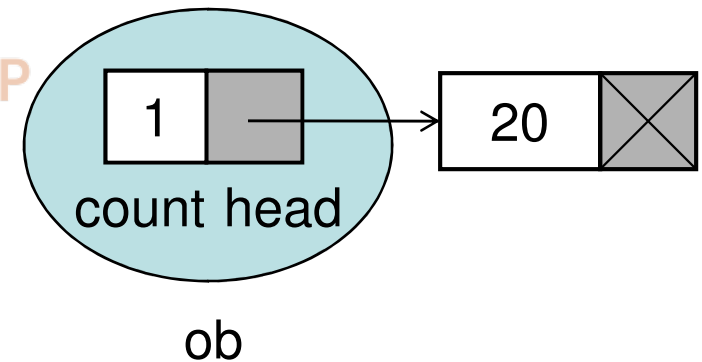
    int InsertFirst(List_ItemType value);
    int InsertLast(List_ItemType value);
    int InsertItem(List_ItemType value, int Position);
    List_ItemType DeleteFirst();
    List_ItemType DeleteLast();
    int DeleteItem(int Position);
    int GetItem(int Position, List_ItemType &dataOut);
    void Print2Console();
    // Augment more methods for linked list here!!!
    LinkedList<List_ItemType>* Clone();
protected: // as previous slide
```

# Pointer vs. Object Variable

```
void main() {  
    LinkedList *p = new LinkedList();  
    p->InsertLast(20);  
    // do sth with p here  
    func(p);  
    delete p;  
}
```

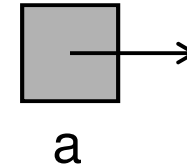


```
void main() {  
    LinkedList ob;  
    ob.InsertLast(20);  
    // do sth with ob here  
    func(ob);  
}
```

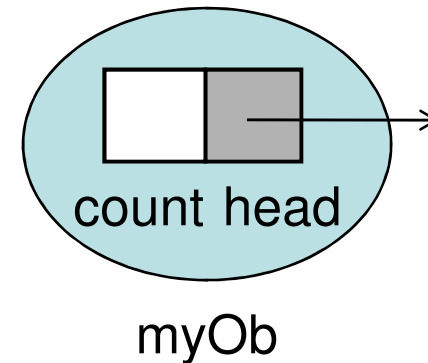


# Pointer vs. Object Variable

```
void func(LinkedList *a)
{
    a->InsertFist(10);
}
```



```
void func(LinkedList myOb)
{
    myOb.InsertFist(10);
}
```



❑ What are the pros and cons?

# Sample Solution: Insert

---

```
template <class List_ItemType>
int LinkedList<List_ItemType>::InsertItem(List_ItemType value,
    int position) {
    if (position < 0 || position > this->count)
        return 0;
    Node<List_ItemType>* newPtr, *pPre;
    newPtr = new Node<List_ItemType>();
    if (newPtr == NULL)
        return 0;
    newPtr->data = value;
    if (head == NULL) {
        head = newPtr;
        newPtr->next = NULL;
    }
    else if (position == 0) {
        newPtr->next = head;
        head = newPtr;
    }
}
```

# Sample Solution: Insert

---

```
else {  
    // Find the position of pPre  
    pPre = this->head;  
    for (int i = 0; i < position-1; i++)  
        pPre = pPre->next;  
    // Insert new node  
    newPtr->next = pPre->next;  
    pPre->next = newPtr;  
}  
this->count++;  
return 1;  
}
```

# Sample Solution: Delete

```
template <class List_ItemType>
int LinkedList<List_ItemType>::DeleteItem(int position){
    if (position < 0 || position > this->count)
        return 0;
    Node<List_ItemType> *dltPtr, *pPre;
    if (position == 0) {
        dltPtr = head;
        head = head->next;
    } else {
        pPre = this->head;
        for (int i = 0; i < position-1; i++)
            pPre = pPre->next;
        dltPtr = pPre->next;
        pPre->next = dltPtr->next;
    }
    delete dltPtr;
    this->count--;
    return 1;
}
```



# Sample Solution: Clone

---

```
template <class List_ItemType>
LinkedList<List_ItemType>*
    LinkedList<List_ItemType>::Clone() {
    LinkedList<List_ItemType>* result =
        New LinkedList<List_ItemType>();
    Node<List_ItemType>* p = this->head;
    while (p != NULL) {
        result->InsertLast(p->data);
        p = p->next;
    }
    result->count = this->count;
    return result;
}
```

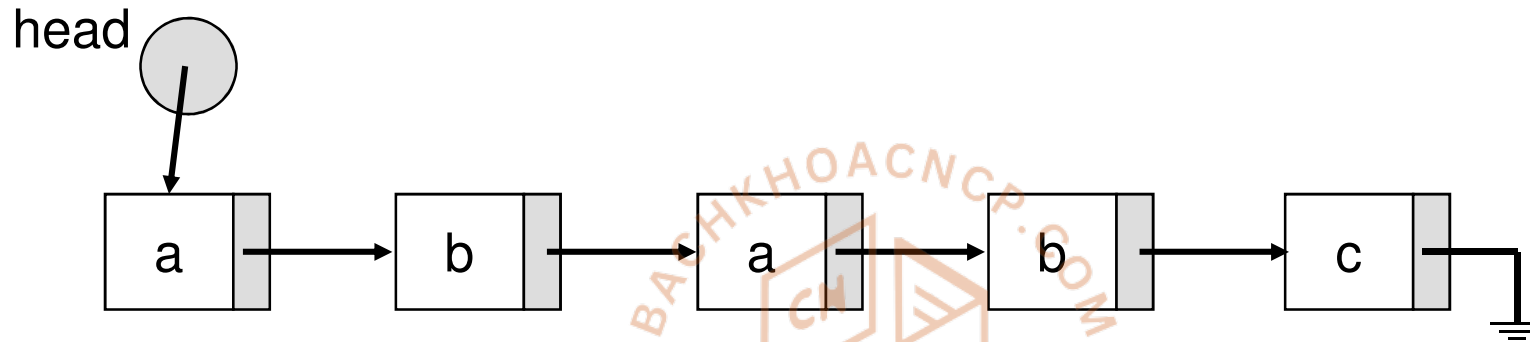
BACH KHOA CNCP.COM

TÀI LIỆU SƯU TẬP

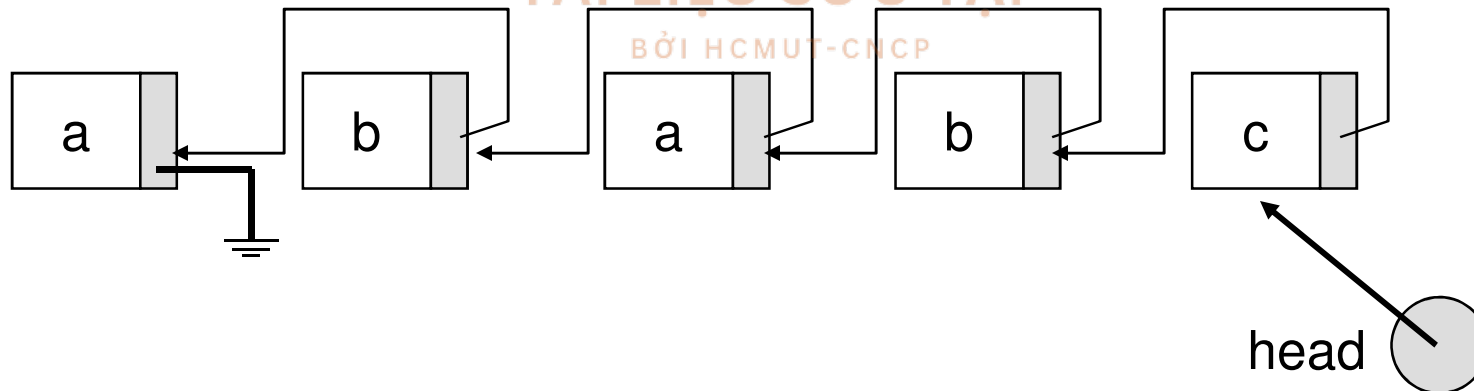
BỞI HCMUT-CNCP

# Homework

## ❑ Reverse a linked list

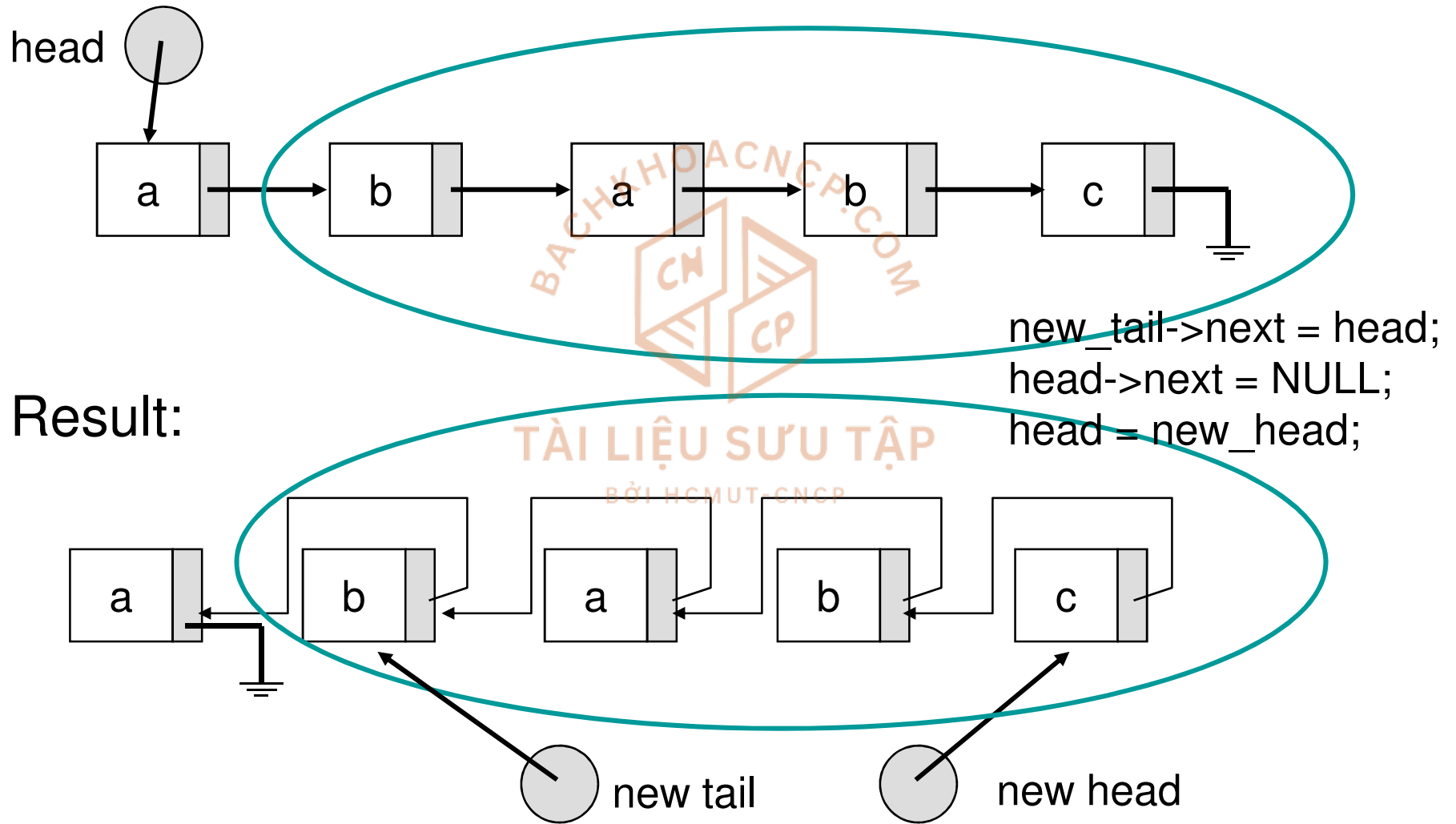


Result:



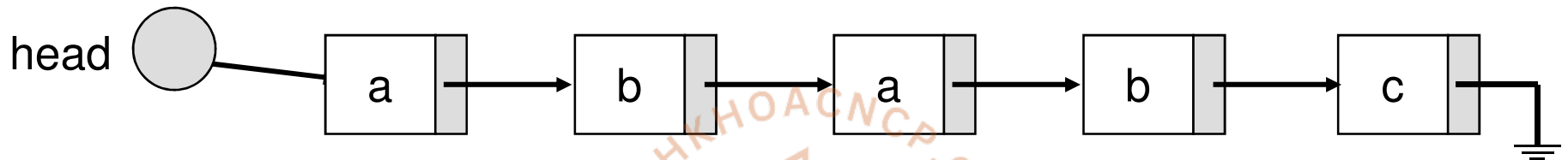
# Homework

## Hint 1

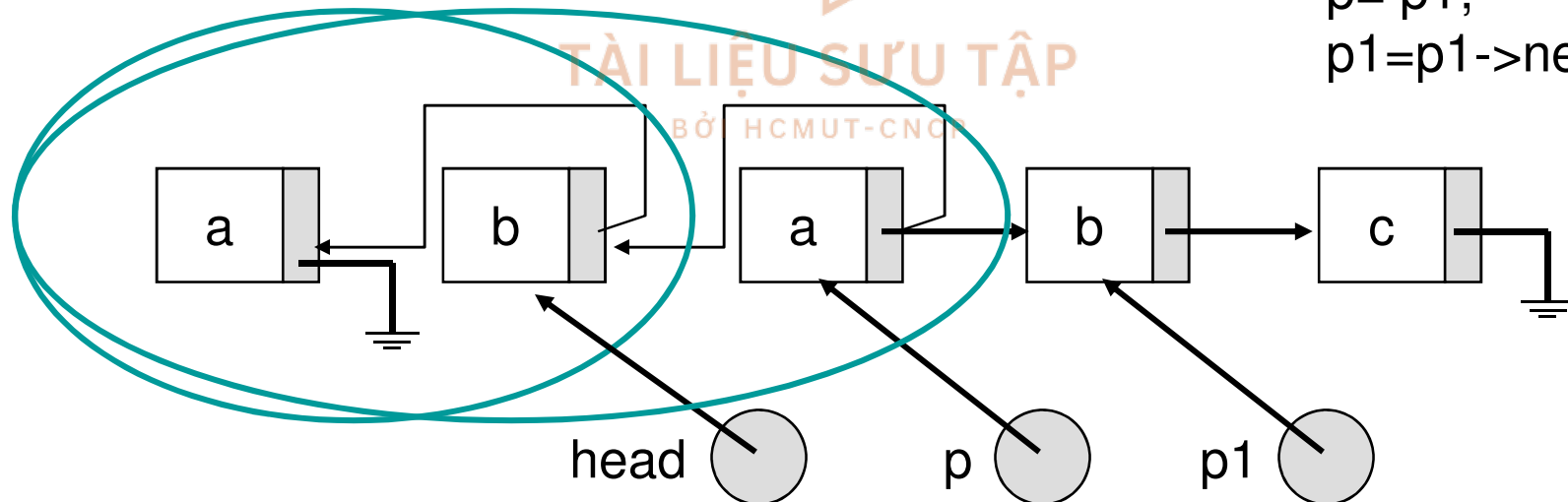


# Homework

## Hint 2



Result:



```
p->next = head;  
head = p;  
p = p1;  
p1 = p1->next
```

# Homework

---

```
template <class List_ItemType>
void LinkedList<List_ItemType>::Reverse() {
    ...
}
```

